2013

Distinct random sampling from a distributed stream

Yung-Yu Chung
Iowa State University

Distinct random sampling from a distributed stream

by

Yung-Yu Chung

A thesis submitted to graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Srikanta Tirthapura, Major Professor

Suraj Kothari

Pavan Aduri

Tien N. Nguyen

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We consider continuous maintenance of a random sample of distinct elements from a massive data stream, whose input elements are observed at multiple distributed sites that communicate via a central coordinator. At any point, when a query is received at the coordinator, it responds with a random sample from the set of all distinct elements observed at the different sites so far. We present the first algorithms for distinct random sampling on distributed streams. We also present a lower bound on the expected number of messages that must be transmitted by any distributed algorithm, showing that our algorithm is message optimal to within a factor of four. We present extensions to sliding windows, and detailed experimental results showing the performance of our algorithm on real-world data sets.

CHAPTER 1. Introduction

Random sampling is a flexible and general method for constructing a synopsis of a data stream. A random sample can be used to answer aggregate queries approximately, with provable, but probabilistic guarantees on the quality of approximation.

We consider the maintenance of a *distinct sample*, a random sample of all distinct elements within the stream. A distinct sample has the property that the probability of an element's inclusion in the sample is independent of the frequency of the element in the population, this property is useful in computing certain aggregates on data. Such a sample can provide estimates for a query pertaining to the set of distinct elements within the stream. For instance, a common query of such a form is simply the number of distinct elements in the stream. A distinct sample can go beyond answering simple distinct count queries, by answering queries about subsets of elements in the stream, for instance, "how many distinct visitors have used a web service, who come from a particular country?", or "what is the average age of the distinct users of this website"? In general, such a sample can be used in determining an aggregate on the set of distinct elements in the stream that satisfy a given predicate, where the predicate itself is supplied only at query time.

We consider the maintenance of a distinct sample over a distributed stream, whose elements are not all observed at the same processor. We adopt the *continuous distributed monitoring* model Cormode (2013); Cormode et al. (2011, 2012); Tirthapura and Woodruff (2011); Woodruff and Zhang (2012), where a "coordinator" node is required to continuously maintain an aggregate function over the union of all streams observed at the different distributed sites. For distributed processing of large data sets, the bottleneck is often the network bandwidth rather than processing speed or memory, and hence the focus is usually on minimizing the communication complexity of the streaming algorithm.

There has been a long line of research on random sampling in the streaming model, starting from the popular *reservoir sampling* algorithm, due to Waterman (see Algorithm R from Vitter (1985)), which has been known since the 1960s. This includes work on speeding up reservoir sampling Vitter (1985), weighted reservoir sampling Efraimidis and Spirakis (2006, 2008), sampling over a sliding window and stream evolution Aggarwal (2006); Babcock et al. (2002); Braverman et al. (2012); Gemulla and Lehner (2008), and distributed random sampling Cormode et al. (2012); Tirthapura and Woodruff (2011). While there is prior work on distinct sampling over a single data stream Frahling et al. (2005); Ganguly et al. (2004); Gibbons (2001); Gibbons and Tirthapura (2001), none of the above consider distinct sampling over a distributed stream.

Our Contributions We present the first distributed algorithms for maintaining a distinct sample in the continuous distributed model. We also present a lower bound on the message complexity of *any* algorithm for this problem, showing that our algorithm has *optimal* message complexity to within constant factors. Let k denote the number of distributed sites, s the desired sample size. *The expected message complexity of maintaining a distinct sample when a total of d elements are observed is* $\Theta\left(ks \ln \frac{de}{s}\right)$.

We show how our algorithm can be extended to support time-based sliding windows over distributed streams. We then present detailed experimental results evaluating the performance of our algorithms over real-world data sets.

It is interesting to compare the message complexity of distributed distinct sampling (DDS) to that distributed random sampling (DRS) from the set of all elements observed by the system, and where element frequency does matter. From previous work Cormode et al. (2012); Tirthapura and Woodruff (2011), we know that the message complexity of DRS for n total elements and k sites is $\Theta\left(\frac{k \log(n/s)}{\log(k/s)}\right)$ if $s < k/8$ and $\Theta(s \log(n/s))$ if $s \geq k/8$. Note that these expressions are tight (up to constant factors), due to the presence of matching lower bounds.

In case of infinite windows, the cost of DDS increases as the product of k, the number of sites, and s, the sample size, while the cost of DRS increases (approximately) as $\max\{k, s\}$.

Thus surprisingly, the message cost of DDS is inherently larger than that of DRS if $\log(d/s)$ is comparable to $\log(n/s)$.

With DRS, when n elements are received in the system, the probability of a new element being selected into the sample decreases as s/n. In case of DDS, the probability of a new element being selected into the sample decreases as s/d, where d is the number of distinct elements in the system, but due to the possibility of the same element appearing at different sites, more messages may be communicated between the sites and the coordinator. Our analysis shows that greater coordination is inherently necessary for DDS than for DRS.

Related Work

There is a growing literature on algorithms for continuous distributed monitoring including the basic *countdown problem* Cormode et al. (2008) frequency moments Cormode et al. (2008); Woodruff and Zhang (2012), entropy Arackaparambil et al. (2009), heavy-hitters and quantiles Yi and Zhang (2013), and often, matching lower bounds Woodruff and Zhang (2012). For a recent survey on results in this model, see Cormode (2013). A survey of sampling on streams appears in Lahiri and Tirthapura (2009).

A geometric approach to distributed monitoring is presented by Giatrakos et al. (2012); Sharfman et al. (2007). This approach breaks down the distributed monitoring of a function into monitoring several local properties. We are not aware of a specialization of this geometric technique to the problem of random sampling. A related model of distributed streams was considered in Gibbons and Tirthapura (2001, 2002). In this model, the coordinator was not required to continuously maintain an aggregate, but instead, when the query was posed to the coordinator, the sites would be contacted and the query result would be constructed. In their model, the coordinator could be said to be "reactive", whereas in the model considered in this paper, the coordinator is "pro-active".

Roadmap: We first present the model and problem definition in Section 2, the algorithm and lower bound for the infinite window case in Section 3, the algorithm and lower bound for the sliding window case in Section 4, and an experimental evaluation in Section 5.

CHAPTER 2. Model

We consider a system with k sites, numbered from 1 to k. Each site i monitors a local stream of elements, which are not all necessarily distinct. There is an integer time associated with each observation, and time is non-decreasing within a stream. At time t, let $\mathcal{S}_i(t)$ denote the stream observed by site i so far, and let $\mathcal{S}(t) = \cup_{i=1}^{k} \mathcal{S}_i(t)$ denote the stream observed by the system so far. Let $\mathcal{D}(t)$ denote the set of distinct elements in $\mathcal{S}(t)$, $n(t)$ the number of elements in $\mathcal{S}(t)$, and $d(t)$ the number of distinct elements in $\mathcal{S}(t)$.

There is a *coordinator* node, different from the other k sites, to whom all queries are posed. The different sites as well as the coordinator are assumed to be synchronized in time, and we ignore message delay from the site to the coordinator. These assumptions, which have also been used in previous works in the continuous distributed streaming model Cormode et al. (2011, 2010); Tirthapura and Woodruff (2011), allow us to focus on communication efficiency.

We consider two versions, based on the scope of the data that the query addresses. In the *infinite window* case, at every time instant t, the coordinator must maintain a random sample of size $\min\{s, d\}$ from $\mathcal{D}(t)$. In the *sliding window* case, we are given a window size w as a parameter, and the user is interested in all elements that have arrived in the most recent w time intervals. In particular, let $\mathcal{S}_i^w(t)$ denote the stream that has arrived at site i at times $t - w + 1, t - w + 2, \ldots, t$, and let $\mathcal{S}^w(t) = \cup_{i=1}^{k} \mathcal{S}_i(t)$. Let $\mathcal{D}^w(t)$ denote the set of distinct elements in in $\mathcal{S}^w(t)$. The goal is for the coordinator to maintain at all times t, a random sample of size $\min\{s, d\}$ from the elements in $\mathcal{D}^w(t)$.

Our measure of performance is the total number of messages sent between the coordinator and the sites, and we aim to minimize this number. Since each message in our algorithm is of a small size, this is also a measure of the number of bytes transmitted, in our case. The size [1] of

[1] The message size is constant, assuming that each stream element can be stored in a constant number of

local data stream S_i, the order of arrival, the number of distinct elements in the local stream, and the interleaving of the stream at different sites, can all be arbitrary, and the algorithm cannot make any assumption about these.

bytes.

CHAPTER 3. Infinite Window

We first consider the case of infinite windows, when the sample has to be chosen from the set of all distinct elements seen so far in the stream. Let $h : \mathcal{U} \to [0, 1]$ be a hash function that assigns a real number in the range $[0, 1]$ to each element in $\mathcal{U}$. For different inputs, it is assumed that the outputs of h are mutually independent random variables. For set S, let $h(S)$ denote $\{h(x)|x \in S\}$. Note that for any time t, $h(\mathcal{S}(t)) = h(\mathcal{D}(\mathcal{S}(t)))$. The basic sampling strategy is as follows. *The distinct sample at time t is the set of elements from $\mathcal{S}(t)$ that yield the s smallest elements in $h(\mathcal{S}(t))$.*

It is clear that the above yields a distinct sample from $\mathcal{S}(t)$. To see this, Take any subset $T \subseteq \mathcal{D}(\mathcal{S}(t))$ of size s; the probability that the elements in T will yield the s smallest values in $h(\mathcal{D}(\mathcal{S}(t)))$ is exactly the probability that in a random permutation of $h(\mathcal{D}(\mathcal{S}(t)))$, the elements in T are ordered before the rest, which is $1/\binom{|\mathcal{D}(t)|}{s}$.

Our distributed algorithm for maintaining the above sample is as follows. Let $u(t)$ denote the value of the sth smallest element in $h(\mathcal{S}(t))$. The coordinator always has the current value of $u(t)$. Each site i maintains a state variable $u_i(t)$, which is its local view of $u(t)$. $u_i(t)$ is initialized to 1 and is updated as follows. Whenever site i observes an item e such that $h(e) < u_i(t)$, e and $h(e)$ are sent to the coordinator, who updates $u(t)$. If $h(e)$ indeed changed the value of $u(t)$, then e is selected into the sample at the coordinator, replacing a current element in the sample (unless fewer than s distinct elements have been seen so far). In turn, the coordinator sends a message back to i to refresh the value of $u_i(t)$. The algorithm at site i is presented in Algorithm 1 and at the coordinator is in Algorithm 2.

Algorithm 1: Infinite Window: Algorithm at site i

Initialization: Receive hash function h from the coordinator; $u_i \leftarrow 1$
repeat
  When site receives element e**: if** $h(e) < u_i$ **then**
    Send e to the Coordinator
    Receive u' from the Coordinator
    $u_i \leftarrow u'$
  end
until *Forever*

Algorithm 2: Infinite Window: Algorithm at the Coordinator

/* P is the random sample, and variable u has the current value of $u(t)$. */
Initialization: $P \leftarrow \phi$, $u \leftarrow 1$
repeat
  if *receive* e *from site* i **then**
    if $h(e) < u$ **then**
      If $e \notin P$, then insert e into P
      if $|P| > s$ **then**
        Discard element e' from P with the largest value of $h(e')$
        Update $u \leftarrow \max\{h(f) | f \in P\}$
      end
    end
    Send u to site i;
  end
  if *a query for a random sample arrives* **then**
    Send P
  end
until *Forever*

Proof of Correctness

Lemma 1. *When queried at time t, the coordinator returns a random sample of size* $\min\{s, d\}$ *selected without replacement from* $\mathcal{D}(t)$.

Proof. First, note that the variable u_i tracks $u_i(t)$ at every time instant t. Further, for each site, $u_i \geq u$, since each time u_i is changed, it is set to u. Since u is non-increasing, it is always true that $u_i \geq u$. Assuming that the hash outputs for different elements are distinct, we can verify that the value of u is equal to the ℓth smallest hash value seen so far, where $\ell = \min\{s, d\}$. Further we can verify that $\mathcal{P}$ contains those elements with the ℓ smallest hash values. This constitutes a random sample of size s chosen without replacement from $\mathcal{D}(t)$. □

3.1 Infinite Window: Analysis

We present an analysis of the message complexity of our algorithm. In Section 3.1.0.1, we present an upper bound on the message complexity of our algorithm and a lower bound is derived in Section 3.1.0.2.

3.1.0.1 Upper Bound

Consider the execution of the algorithm until the end of time step t. Let $d = |\mathcal{D}(t)|$ be the total number of distinct elements that were observed in the stream. Let d_i denote the total number of distinct elements in stream $\mathcal{S}_i(t)$ observed at site i. Clearly, $d_i \leq d$. Let Y_i denote the total number of messages that are sent and received by node i (note that the number of messages sent by site i equals the number of messages received). Let Y denote the total number of messages in the system. In the following, when the context is clear, we use $\mathcal{S}_i$ to mean $\mathcal{S}_i(t)$, $\mathcal{D}_i$ to mean $\mathcal{D}_i(t)$, and so on.

We first observe that for any $e \in \mathcal{S}_i$, site i does not incur any communication cost for repeated occurrences of e; $h(e)$ cannot be less than u_i for such repeat occurrences. For $j = 1 \ldots d_i$, let e_i^j be the jth new distinct element in the local stream $\mathcal{S}_i$, for example, e_i^2 is the second distinct element in $\mathcal{S}_i$, and so on. For $j = 1 \ldots d_i$, let Y_i^j be a random variable equal to 1 if site i communicated with the coordinator upon receiving e_i^j, and 0 otherwise. Since

each communication from site i to the coordinator also results in a return message from the coordinator, we have:

$$Y_i = 2\sum_{j=1}^{d_i} Y_i^j \tag{3.1}$$

Lemma 2. *For site i, and $j = 1 \ldots d_i$, $\Pr\left[Y_i^j = 1\right] \leq s/j$, if $j > s$.*

Proof. Let Z_i^j denote that event that $h(e_i^j)$ is among the s smallest elements in $\{h(e_i^q) | q = 1 \ldots j\}$. Note that if $Y_i^j = 1$, then Z_i^j must be true. Note that the converse is not necessarily true; it is possible that Z_i^j is true, but $Y_i^j = 0$, for example, e_i^j may have already been observed by a site other than i, and its value may have been incorporated into u, and hence into u_i. It is easy to see that $\Pr\left[Z_i^j\right] = \frac{s}{j}$. Since $\Pr\left[Y_i^j = 1\right] \leq \Pr\left[Z_i^j\right]$, the lemma follows. □

Lemma 3.

$$\mathbb{E}\left[Y_i\right] \leq 2s + 2s\left(H_{d_i} - H_s\right)$$

Proof. Using Equation 3.1 and linearity of expectation:

$$\begin{aligned}
\mathbb{E}\left[Y_i\right] &= 2\sum_{j=1}^{d_i} \mathbb{E}\left[Y_i^j\right] = 2\sum_{j=1}^{d_i} \Pr\left[Y_i^j = 1\right] \\
&= 2\sum_{j=1}^{s} \Pr\left[Y_i^j = 1\right] + 2\sum_{j=s+1}^{d_i} \Pr\left[Y_i^j = 1\right] \\
&\leq 2s + 2\sum_{j=s+1}^{d_i} \frac{s}{j} = 2s + 2s\left(H_{d_i} - H_s\right)
\end{aligned}$$

where we have used Lemma 2. □

Lemma 4. *Let Y denote the number of messages transmitted by the distributed algorithm during an execution when d distinct elements are observed overall.*

$$\mathbb{E}\left[Y\right] \leq 2ks + 2ks\left(H_d - H_s\right) \approx 2ks\left(1 + \ln\left(\frac{d}{s}\right)\right)$$

Proof. The proof follows from the observation $Y = 2\sum_{i=1}^{k} Y_i$, combined with Lemma 3, and the observation $d_i \leq d$. □

We remark that using Lemma 3, it is possible to get a tighter upper bound for $\mathbb{E}[Y]$ in cases when the numbers of distinct elements observed at individual sites is much smaller than the number of distinct elements overall.

Observation 1.

$$\mathbb{E}[Y] \leq 2ks + 2s\sum_{i=1}^{k}(H_{d_i} - H_s) \approx 2ks + 2s\sum_{i=1}^{k}\ln\left(\frac{d_i}{s}\right)$$

The following theorem summarizes the performance of the algorithm for infinite windows.

Theorem 1. *Let d, s, and k respectively denote the total number of distinct elements in the distributed stream, sample size, and the number of sites. There is an algorithm that continuously maintains a distinct sample of a distributed stream, whose expected total number of messages is $O(ks\ln(\frac{de}{s}))$, memory consumption per site is $O(1)$, memory consumption at the coordinator is $O(s)$, and processing time per element is $O(1)$.*

3.1.0.2 Lower Bound

Given any distributed algorithm $\mathcal{A}$ that continuously maintains a distinct sample of the stream of size s, we construct an input which causes $\mathcal{A}$ to send at least a certain minimum number of messages, in expectation. Suppose that the elements were all chosen from the set $[m] = \{1, 2, \ldots, m\}$ for $m >> k$.

In showing that there must be a minimum amount of communication in a distributed protocol, we have to first deal with the fact that in a synchronous distributed system such as the one we are assuming, two processors can communicate without actually sending a message; for example, the absence of a message from a site in a given round already conveys information to the coordinator, if for a different input in the same round, the site did send a message to the coordinator.

Lemma 5. *For any algorithm $\mathcal{A}$, and any site $i = 1 \ldots k$, there cannot be more than one element $e_i \in [m]$ such that upon receiving a stream with only e_i, site i sends a message with probability less than* 0.25.

Proof. Suppose that there were two such elements e_i and e_i' such that upon observing either element, site i sent a message with probability less than 0.25, according to algorithm $\mathcal{A}$. Consider two inputs $\mathcal{I}_1$ and $\mathcal{I}_2$, which did not assign any elements to sites 2 till k, and assigned e_i and e_i' respectively to site 1. With probability at least 0.5, site 1 will not send a message to the coordinator for either input e_i or e_i'.

Thus with probability at least 0.5, the coordinator's view of site 1 is the same for both inputs. Similarly, the input streams at all other nodes are identical in $\mathcal{I}_1$ and $\mathcal{I}_2$, leading to an identical distribution of the other sites, in the coordinator's view. However, with $\mathcal{I}_1$, the random sample must be e_i, and with $\mathcal{I}_2$, the random sample must be e_i'. With probability at least 0.5, the coordinator must make an error in the random sample, which is a contradiction since the coordinator must have a random sample at all times. □

Lemma 6. *For any algorithm $\mathcal{A}$, there is an input $\mathcal{I}^1$ that sends exactly one element, e^1, to each of the k sites, and the expected number of messages sent is at least $\frac{k}{4}$.*

Proof. From Lemma 5, for each site i, there exists an element e_i such that if site i received an element $e \neq e_i$, then the expected number of messages sent by site i is at least $1/4$. Choose an arbitrary element $e^1 \in [m] - \cup_{i=1}^{k}\{e_i\}$. The input $\mathcal{I}^1$ is constructed by assigning e^1 to each site $i = 1 \ldots k$. Each site i sends an expected at least $1/4$ messages on this input, and the expected total number of messages is at least $\frac{k}{4}$. Note that we have used the fact $m >> k$. □

Lemma 7. *Suppose a set $\mathcal{D}$ of distinct elements of size d have already been observed by the system so far, after some rounds of computation. For any algorithm $\mathcal{A}$, and any site $i = 1 \ldots k$, there is an element $e_i^{\mathcal{D}}$ such that upon receiving any element $e \in [m] - e_i^{\mathcal{D}} - \mathcal{D}$ in the next round, site i will send a message to the coordinator with probability at least $\frac{s}{2(d+1)}$.*

Proof. Suppose that the set $\mathcal{D}$ has already been observed by the system so far. Suppose that there were two distinct elements $e_i, e_i' \in [m] - \mathcal{D}$ such that upon e_i, the probability that i sent a message to the coordinator was less than $\frac{s}{2(d+1)}$, and also upon e_i', the probability that i sent a message to the coordinator was less than $\frac{s}{2(d+1)}$.

Consider the following two inputs in the next round. In one input $\mathcal{I}^1$, site i is given e_i and the other sites do not receive any element. In the other input, site i is given e_i' and the other

sites do not receive an element. In $\mathcal{I}^1$, at the end of this round, there is a probability of $\frac{s}{d+1}$ that e_i belongs to the random sample. In $\mathcal{I}^2$, at the end of this round, there is a probability of $\frac{s}{d+1}$ that e_i' belongs to the random sample. Thus, with probability at least $\frac{s}{d+1}$, the sample at the coordinator after observing $\mathcal{I}^1$ is different from the sample after observing $\mathcal{I}^2$.

However, in this round, the coordinator observes a change in execution between $\mathcal{I}^1$ or $\mathcal{I}^2$ only if either (1) site i sends a message to the coordinator in $\mathcal{I}^1$, or (2) site i sends a message tot he coordinator in $\mathcal{I}^2$. The probability that one of the above events happen is less than $\frac{s}{d+1}$; and further the behavior of the other sites has an identical distribution in both inputs, since they did not receive any element.

Thus, we have that the contents of the random sample at the end of the round are different with probability at least $\frac{s}{d+1}$, but the probability that the messages observed by the coordinator are different is less than $\frac{s}{d+1}$. This leads to a non-zero probability that the random sample at the coordinator is incorrect at the end of this round, and hence a contradiction.

Hence, there can be at most one element e_i such that upon receiving an element $e \in [m] - e_i - \mathcal{D}$, the probability of sending a message to the coordinator is at least $\frac{s}{2(d+1)}$, and the lemma follows. □

Lemma 8. *Suppose the set of distinct elements observed so far is $\mathcal{D}$, and let d be the size of $\mathcal{D}$, and $d \geq s$. For any algorithm $\mathcal{A}$, there exists another round of input $\mathcal{I}(\mathcal{D})$ such that after observing $\mathcal{I}(\mathcal{D})$, the sites will send at least an expected $\frac{ks}{2(d+1)}$ elements to the coordinator.*

Proof. The input $\mathcal{I}(\mathcal{D})$ is constructed as follows. Let e be any element such that $e \notin \mathcal{D} \cup \left(\cup_{i=1}^{k}\{e_i^{\mathcal{D}}\}\right)$. Element e is given to every site in this round. Using Lemma 7, we get that each site $i = 1 \ldots k$ will send a message to the coordinator with probability at least $\frac{s}{2(d+1)}$. The lemma follows. □

Lemma 9. *For any algorithm $\mathcal{A}$, there exists an input distributed stream, $\mathcal{I}_{\mathcal{A}}$ with d distinct elements such that the expected number of messages sent by the algorithm upon receiving $\mathcal{I}_{\mathcal{A}}$ is at least $\frac{ks}{2}\left(H_d - H_s + 1\right) \approx \frac{ks}{2}\ln\left(\frac{de}{s}\right)$.*

Proof. Input $\mathcal{I}_{\mathcal{A}}$ is constructed as follows. Let $\mathcal{D}_0 = \phi$. The input in the first round is $\mathcal{I}(\mathcal{D}_0)$.

For $i \geq 0$, let the set of all distinct elements observed till (and including) round i be $\mathcal{D}_i$. Note that the size of $\mathcal{D}_i$ is exactly i. For $i = 0 \ldots (d-1)$, the input in round $i+1$ is $\mathcal{I}(\mathcal{D}_i)$.

From Lemma 8, in round $i > s$, the expected number of messages sent to the coordinator is at least $\frac{ks}{2(i+1)}$. Summing this over all rounds $s+1, \ldots, d$, we get the number of messages to be at least $\frac{ks(H_d - H_s)}{2}$.

For rounds 1 till s, similar methods yield that the expected number of messages sent in each round must be at least $\frac{k}{2}$, for the above input. Thus the expected total number of messages sent by this algorithm should be at least $\frac{ks}{2}(H_d - H_s + 1)$. □

Sampling With Replacement Thus far, we have considered distinct sampling without replacement. In sampling with replacement, the s different samples are all chosen independently and randomly from the set of distinct elements observed so far, $\mathcal{D}(t)$.

One solution to distinct sampling with replacement is to repeat s parallel copies of the single element sampling algorithm, each copy using a different hash function. The correctness of this scheme is trivial, and the message cost is s times the cost of a single element sampling algorithm, which is $O(sk \log de)$. Comparing with the cost of distinct sampling without replacement $O(ks \log(de/s))$, the message cost of this algorithm for sampling with replacement is close to the message cost of sampling without replacement.

We also note there is an easy reduction from distinct sampling without replacement to distinct sampling with replacement; from a distinct sample with replacement of size slightly greater than s; it is easy to prove that the result is indeed a distinct sample without replacement from the original dataset. Thus, the lower bound of $O(ks \ln(\frac{de}{s}))$ applies to both sampling with and without replacement, and this simple method leads to an algorithm for sampling with replacement with near-optimal message complexity.

CHAPTER 4. Sliding Windows

We now consider the sliding windows version of the problem. Let k denote the number of sites, as before, and we assume that time is divided into "slots" where the slots are numbered consecutively in an increasing sequence. At each slot t, a site may receive zero, one, or more elements. It is assumed that time is synchronized across the sites so that when site 1 is observing elements in slot t, other sites are also observing elements in the same slot. Given a window size $w > 0$, the goal is: *When a query is issued to the coordinator at slot t, it returns a random sample chosen from all distinct elements observed is slots $(t - w + 1)$ till t, both endpoints inclusive.* We use the terms "slot" and "time" interchangeably in the following discussion.

4.1 Algorithm

For simplicity, we present the algorithm for the case $s = 1$; the extension to larger sample sizes is straightforward. At time t, let $\mathcal{S}_i(t, w)$ denote the elements that have arrived in slots $(t - w + 1)$ till t at site i. Let $\mathcal{S}(t, w)$ denote the (multiset) union of $\mathcal{S}_i(t, w), i = 1 \ldots k$. Let $\mathcal{D}_i(t, w)$ denote the set of distinct elements in $\mathcal{S}_i(t, w)$, and $\mathcal{D}(t, w)$ the set of distinct elements in $\mathcal{S}(t)$. For a set of elements E, let $h(E) = \{h(e)|e \in E\}$. The high-level algorithm idea is similar to the infinite window case: choose the element with the smallest hash values from among all elements in $\mathcal{D}(t, w)$. Let $u(t, w)$ denote the smallest hash value from $h(\mathcal{D}(t, w))$. The problem with implementing the sliding windows scenario, when compared with the infinite window scenario, is that the value of $u(t, w)$ is not monotonically decreasing. As elements expire from the window, the value of $u(t, w)$ may increase, and keeping track of this needs additional communication.

Intuition An algorithm is as follows. Each site i, at all times, keeps track of the element with the smallest hash value from $\mathcal{D}_i(t, w)$. Whenever this changes, the coordinator is informed of the new distinct sample from $\mathcal{D}_i(t, w)$. Since the coordinator at all times has the element with the smallest hash value from each site, it can maintain the element with the globally smallest hash value from $\mathcal{D}(t, w)$. Any changes to this element are communicated to the coordinator by the site. The message complexity of this algorithm depends on how often the local sample at each site i changes.

Note that the above algorithm used no feedback from the coordinator to the site. We can potentially reduce the communication cost of this algorithm in the following manner. Similar to the case of infinite window, the coordinator maintains a variable u, which has the current smallest hash value in $\mathcal{D}(t, w)$. When it replies back to a site, the coordinator conveys the current value of u to the site. However, note that u may actually increase at the coordinator, due to elements expiring from the window, and this needs also to be conveyed to the sites.

One possibility is: each time the value of u increases, the coordinator broadcasts the new value of u to all nodes – note such an action was not necessary in the infinite window case since in that case, the local view of u at a site was never less than the global value of u. With such a broadcast whenever u increases at the coordinator, we can make sure that the sites communicate in a "safe" manner; i.e. messages that can potentially change the state of the coordinator are sent to the coordinator.

But such a broadcast can be expensive, and we instead use an alternate method as follows. The coordinator replies back with the value of u to a site only when the site communicates with the coordinator. With the value of u, the coordinator also sends a time stamp, which is the time at which the sample expires. If u_i (site i's local view of u) expires, then site i falls back to a view of u that is gotten by observing solely the local stream $\mathcal{D}_i(t, w)$. This ensures that the value of u_i is often synchronized with u at the coordinator; and when it is not synchronized with the coordinator, $u_i \geq u$.

There is one other issue, that of maintaining the locally smallest element within $\mathcal{D}_i(t, w)$. It is known that in general, maintaining the smallest element within a sliding window requires space linear in the window size in the worst case Datar et al. (2002). However, we do not

need to maintain the minima over arbitrary numbers, but need to do so over random numbers. Hence, we can use the idea from priority sampling over sliding windows Babcock et al. (2002) to significantly reduce the space consumption at each site.

For elements e, e' and time t, t', we say that tuple (e, t) dominates tuple (e', t') at site i if $t > t'$ and $h(e) < h(e')$. Let $\tau_i(e)$ denote the most recent time when e was observed at site i. For elements e, e', we say e dominates e' at site i if $(e, \tau_i(e))$ dominates $(e', \tau_i(e'))$. Each site i has a data structure T_i consisting of all elements that could potentially be included within the random sample of distinct elements either now, or in the future. An efficient data structure for T_i can be a *treap* Seidel and Aragon (1996).

The algorithm at the site is presented in Algorithm 3, and at the coordinator in Algorithm 4.

4.2 Sliding Window Analysis

Per-site Space Complexity We show that the expected space complexity at site i at time t is $O(\log |\mathcal{D}_i(t, w)|)$.

Lemma 10. *For site i, the expected size of T_i at time t is no more than H_{M_i}, where M_i is the size of $\mathcal{D}_i(t, w)$, and H_j is the jth Harmonic number.*

Proof. This proof is along the lines of Babcock et al. (2002). T_i only contains those tuples (e, t) that are not dominated by another tuple (e', t') at stream $\mathcal{S}_i$. Let M_i denote the size of $\mathcal{D}_i(t, w)$. Let $e_1, e_2, \ldots, e_{M_i}$ be the distinct elements observed by site i within the current window, in the order of their time of expiry; i.e. e_1 expires first, followed by e_2, and so on. e_1 is present in T_i only if $h(e_1) < h(e_j)$ for each $j = 2 \ldots M_i$. Thus, the probability of e_1 being present in T_i is $1/M_i$. Similarly, the probability of e_2 being present in T_i is $1/(M_i - 1)$ and so on. Proceeding thus, we get that the the expected size of T_i is no more than $H_{M_i} \leq H_M$. □

Using Chernoff bounds, it is possible to also provide a high probability bound on the space consumption at each site i. We omit the details here.

Message Complexity At time t at site i, for $j = (t - w + 1) \ldots t$, let $d_i^j(t)$ denote the number of elements in $\mathcal{D}_i(t, w)$ whose most recent appearance in this window was at time j.

Algorithm 3: Sliding Window: sampling algorithm at site i.

```
/* (e_i, u_i, t_i) is the distinct sample from the current window at site i.  e_i
   is the element, u_i = h(e_i), and t_i is the timestamp at which e_i expires.
   */
/* T_i is the set of all tuples that have a chance of getting selected into
   the sample in the future.                                                  */
```

Initialization: $e_i \leftarrow \phi$, $u_i \leftarrow 1$, $T_i \leftarrow \emptyset$
repeat
  if *Receive an element* e *at time* t **then**
    if $e \in T_i$ **then**
      update timestamp of e in T_i to $(t + w)$
    end
    else
      Insert $(e, t + w)$ into T_i
    end
    Delete all elements (e', t') from T_i that have expired, i.e. $t' < t$.
    Delete all elements (e', t') from T_i that are dominated by another element within T_i.
    if $h(e) < u_i$ **then**
      Send (e, t) to the coordinator
    end
  end
  if *Receive* (e, t) *from the coordinator* **then**
    Set $(e_i, u_i, t_i) \leftarrow (e, h(e), t)$
    Insert (e, t) into T_i
    As in the previous step, delete all elements (e', t') from T_i that have expired. And delete all elements (e', t') from T_i that are dominated by another element from T_i.
  end
  if $t_i < t$ **then**
    Remove expired elements from T_i
    Select (e, t) from T_i with the smallest value of $h(e)$, and set $(e_i, u_i, t_i) \leftarrow (e, h(e), t)$
    Send (e, t) to the coordinator
  end
until *Forever*

Algorithm 4: Sliding Window: Sampling Algorithm at the coordinator

```
/* e* is the distinct sample, and t* is the time at which this sample
   expires.  u* = h(e*).                                              */
repeat
    if Receive (e', t') from site i at time t then
        if (u* > h(e')) or (t* < t) then
            (e*, u*, t*) ← (e', h(e'), t')
        end
        Send (e*, t*) to site i.
    end
    if A query arrives for a sample from D(t, w) then
        Return e*.
    end
until Forever
```

Let $M_i(t)$ denote the size of $\mathcal{D}_i(t,w)$. We note that $\sum_{j=t-w+1}^{t} d_i^j(t) = M_i(t)$.

Let $Y_i(t)$ be a random variable equal to 1 is site i communicated with the coordinator at time t, and 0 otherwise. We note that $Y_i(t) = 1$ if (1) Event 1: the current sample expires at time t, or if (2) Event 2:an incoming distinct element is sampled. Let $b_i(t) = \max_{(t-w+1)\leq t'\leq t} d_i^j(t')$.

Lemma 11.

$$\Pr\left[Y_i(t) = 1\right] \leq \frac{b_i(t)}{M_i(t,w)} + \frac{b_i(t-1)}{M_i(t-1,w)}$$

Proof. At time step t, the likelihood that the expiring element is the the chosen sample is equal to $\frac{d_i^{t-w}(t-1)}{M_i(t-1,w)} \leq \frac{b_i(t-1)}{M_i(t-1,w)}$. Similarly, the probability that an incoming element is chosen into the sample is $\frac{d_i^t(t)}{M_i(t,w)} \leq \frac{b_i(t)}{M_i(t,w)}$. The required probability is the sum of the above two probabilities. □

Lemma 12. *The expected total number of messages sent in the system by the algorithm for maintaining a distinct random sample of a single element over a time-based sliding window of size w, with k sites and T timesteps is $O(kT\frac{b}{M})$ where b is the maximum over all sites i and timesteps t of $b_i(t)$ and M is the minimum over all sites i and timesteps t of $M_i(t)$.*

Proof. Clearly in each timestep at each site, the expected number of messages is bounded by $2b/M$ using Lemma 11. Multiplying by the total number of sites and timesteps we arrive at the above expression. □

CHAPTER 5. Experiments

We present an experimental evaluation of our proposed algorithms. We implemented the algorithms in Java, using the MurmurHash Holub hash function. We used two datasets. The first is an OC48 Internet Traces Dataset CAIDA OC48 Trace Project (2006), which has anonymous traffic traces taken at a US west coast OC48 peering link for a large ISP in 2002 and 2003. We consider the concatenation of the sender's IP address and the receiver's IP address to generate an element. The other is the Enron Email Dataset CALO Project (2009), where an element is again constructed by concatenating the sender's email address and the receiver's email address. A summary the data is shown in Table 5.1. Each data point presented is the average of 50 independent runs.

	# Elements	# Distinct
OC48	42,268,510	4,337,768
Enron	1,557,491	374,330

Table 5.1 The number of elements and distinct elements in OC48 IP and Enron e-mail datasets

Our theoretical analysis was for the worst case, when the input data can be distributed in an arbitrary manner by the adversary. In Section 5.1, we evaluate the impact of data distribution on the performance of the algorithm. In Section 5.2, we compare the performance of our method with an alternate, fairly natural algorithm. In Section 5.3, we evaluate the performance of the sliding window version of our algorithm.

5.1 Impact of Data Distribution

We examine the performance of our algorithm under different distribution methods. In the first method, called "flooding", each incoming element is assigned to every site in the system.

In the second method, called "random", an incoming element is sent to a single site, chosen uniformly at random. In the third method, "round-robin", each element is sent to a single site, and the elements are assigned to sites in a round-robin manner, i.e. the j-th element is monitored by site $(j \mod k) + 1$. This experiment is for 5 sites and a sample size of 10.

From Figure 5.1, we observe that at the beginning of observation, the number of messages increases quickly, since the sample is changing often. As more elements are observed, the probability of a change in the sample decreases and fewer messages are sent. It is clear that the number of messages under flooding is significantly larger than the number of messages under random or round-robin, though the total number of distinct elements seen in both inputs is the same. This scenario is explained by our tighter upper bound $2ks\left(1 + \sum_{i=1}^{k} \ln \frac{d_i}{s}\right)$ (see Observation 1). Note that the number of messages for random and round-robin are so similar that they are nearly indistinguishable in the graph; this is because the average numbers of distinct elements received by each site are close to each other. In following graphs, we only present the random distribution and not round-robin.

Figure 5.2 shows the number of messages as a function of the sample size; the message complexity increases almost linearly with the sample size, though the slopes are different for different methods of data distribution. Figure 5.3 shows the number of messages as a function of the number of sites k. For flooding, the number of messages increases linearly with the number of sites. However, for random distribution, the number of messages is much smaller than in case of flooding, and is almost independent of the number of sites.

5.2 Comparison with Other Algorithms

According to our survey, there are no prior methods for distinct sampling on a distributed stream. We compare the performance of our algorithm with a new algorithm, which we call Algorithm "Broadcast". The difference between Algorithm Broadcast and our proposed method is that Algorithm Broadcast will broadcast the current value of u (which has the value of $u(t)$ at time t) to all sites whenever there is an update to u. This version has the advantage that fewer messages are sent from the sites to the coordinator, since the u_is are always in sync with the coordinator. However, this has the downside of requiring a broadcast each time u changes.

We set the number of sites to 100, sample size to 20, and we use the "random" method for data distribution. The results are shown in Figure 5.4. It is clear that Broadcast requires significantly more messages than our algorithm; this suggests that typically it is not worth keeping the different sites synchronized with respect to the value of u. A "lazy" approach of refreshing u when incorrect results in fewer messages.

We also note that the message cost of Algorithm Broadcast is linear in the number of sites k, and the sample size s. However, the slope of the Broadcast algorithm is considerably higher. We show the comparison of the two algorithms as a function of the sample size in Figure **??**. Similar results are observed with the number of sites.

We next consider the influence of the non-uniformity in the streams observed at different sites. Here, we construct a distributed input where one site "dominates" over the other sites in terms of the number of distinct elements that it observes. Each input element is sent to only one site; but instead of dealing it randomly or in a round-robin fashion, we send the element to site 1 with a probability that is a factor α times the probability that a site other than 1 is chosen. We call this factor as the "dominate rate". For example, if the dominate rate were 200, then site 1 is 200 times more likely to receive an element than another site. Figure 5.6 shows the relation between number of message transmissions and the dominate rate for different algorithms. The number of messages transmitted reduces as the dominate rate increases; this is to be expected. Note that the higher the dominate rate, the closer this gets to centralized stream monitoring.

5.3 Sliding Windows

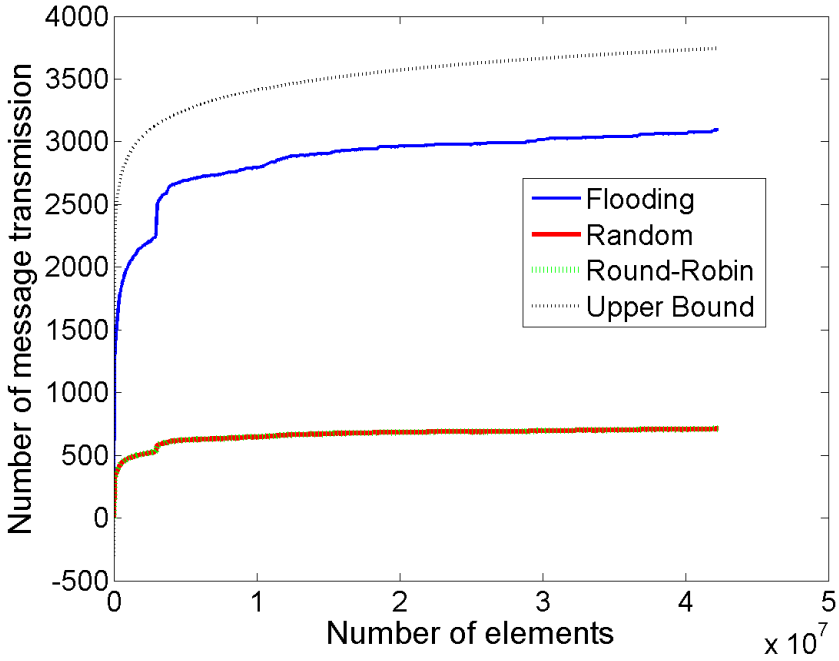
We derive the inputs to sliding windows from the OC48 and Enron Mail datasets as follows. We consider timesteps numbered consecutively from 1 onwards. In each timestep, we assign 5 elements to 5 sites chosen randomly; hence, it is possible that multiple elements are observed by the same site in the same timestep. The memory consumption and communication complexity are recorded at each timestep, for different numbers of sites and window sizes. Each data point is the average of 10 independent experiments.

Impact of Window Size For these experiments, we have fixed the number of sites at 10. Figure 5.7 shows the average memory consumption per site as a function of the window size. Figure 5.8 shows the total number of messages as a function of the window size.

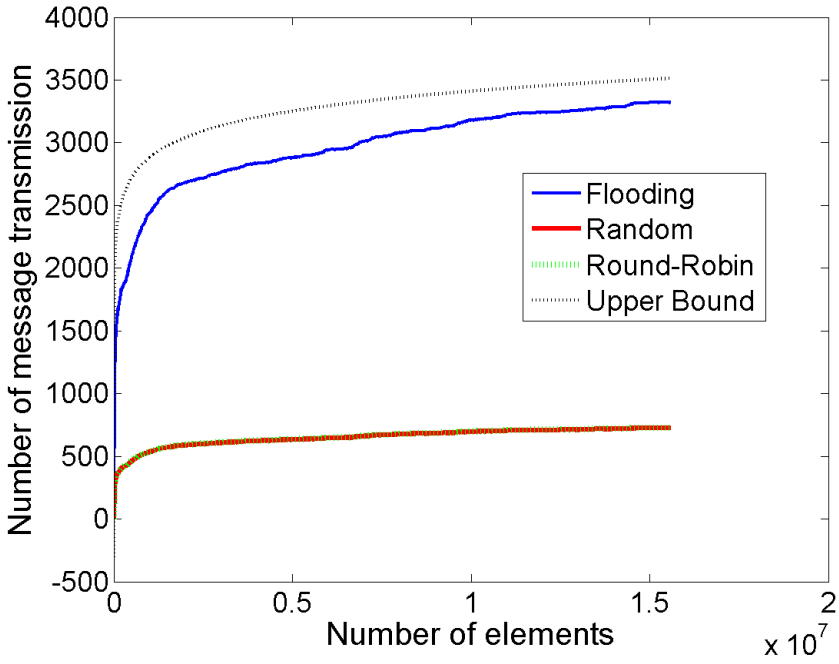
From Figure 5.7, we note that the per site memory consumption is bounded within a region and exceeds the region with a low probability. The memory consumption increases as the window size increases, but the rate of increase decreases with the window size, leading to a logarithmic dependence of the memory on the window size.

From Figure 5.8, we note that unlike the memory consumption, the communication complexity decreases as the window size increases; this ie because with a larger window size, the number of distinct elements within a window increases, and there is a lesser probability of the distinct sample changing due to the arrival of a new element or due to an element expiring from the window.

Impact of Number of Sites Figures 5.9 and 5.10 show the per-site memory and messages respectively when the number of sites k is varying. The window size is fixed at 100. Note that as the number of sites is increased, fewer elements arrive at each site, leading to a lesser memory consumption per site.

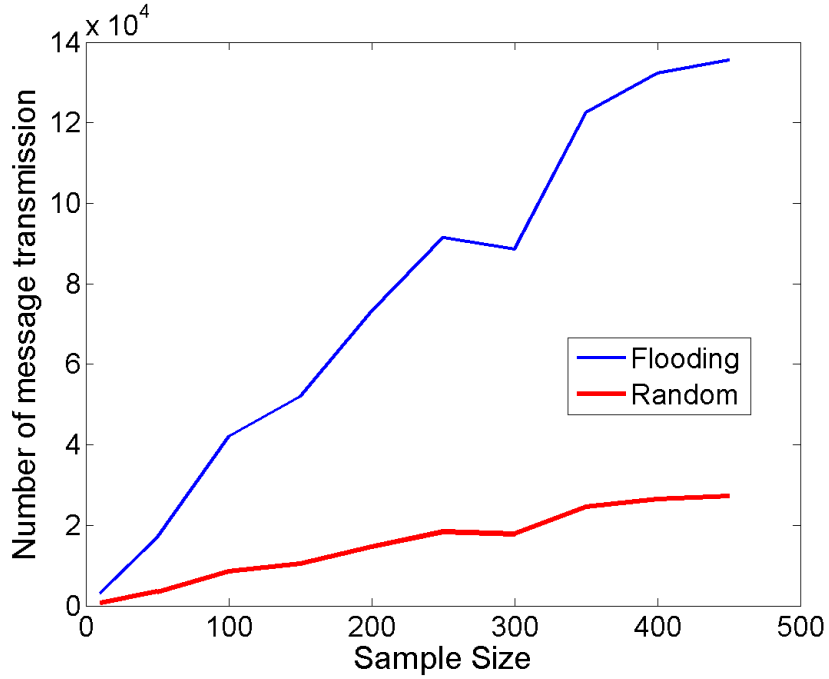


(a) OC48 IP Dataset

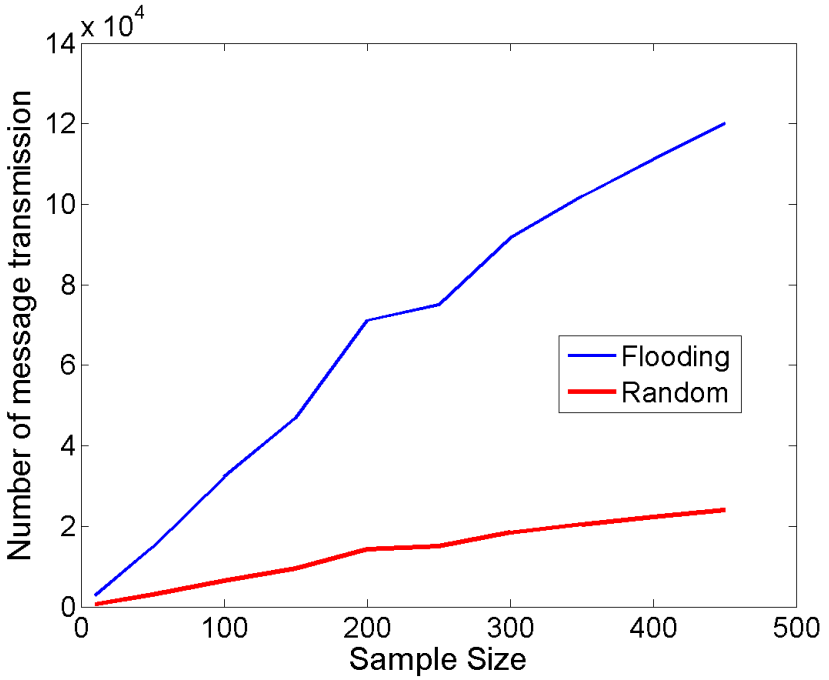


(b) Enron Email Dataset

Figure 5.1 The number of messages under three different methods of data distribution, "flooding", "random", and "round robin". The curves for round-robin and random are nearly identical, so they cannot be differentiated in this picture.

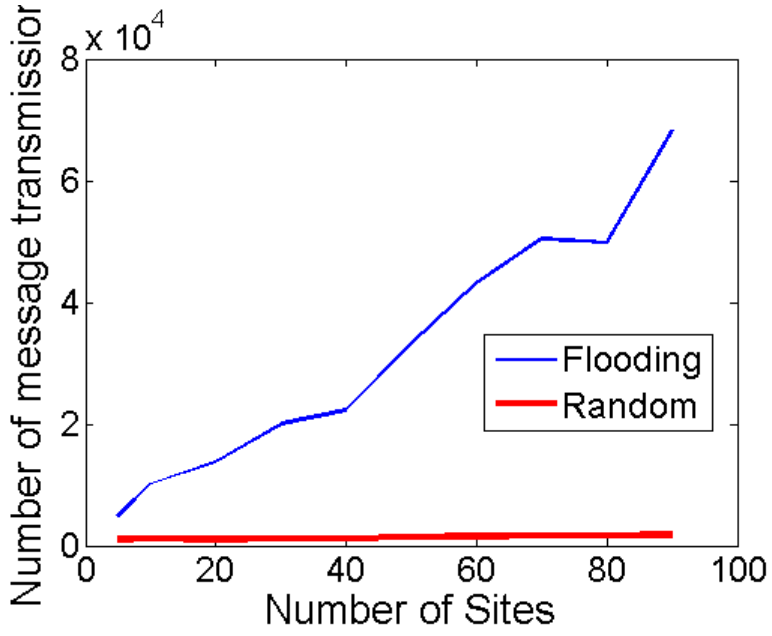


(a) OC48 IP Dataset

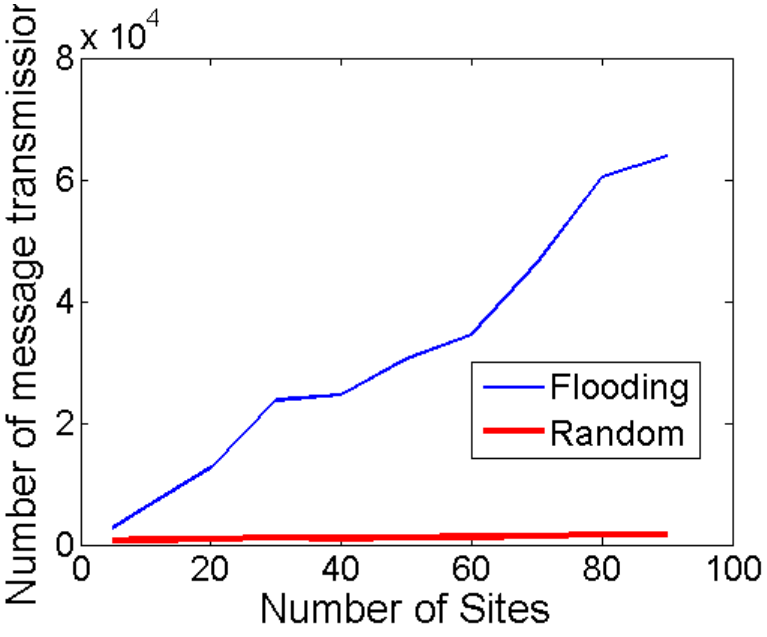


(b) Enron Email Dataset

Figure 5.2 The number of messages as a function of the sample size s.

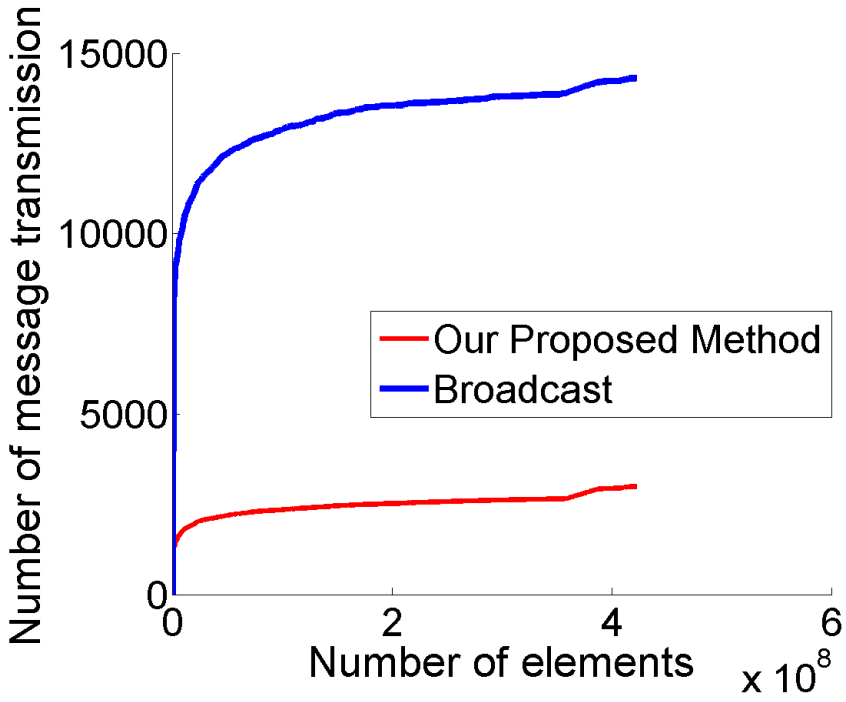


(a) OC48 IP Dataset

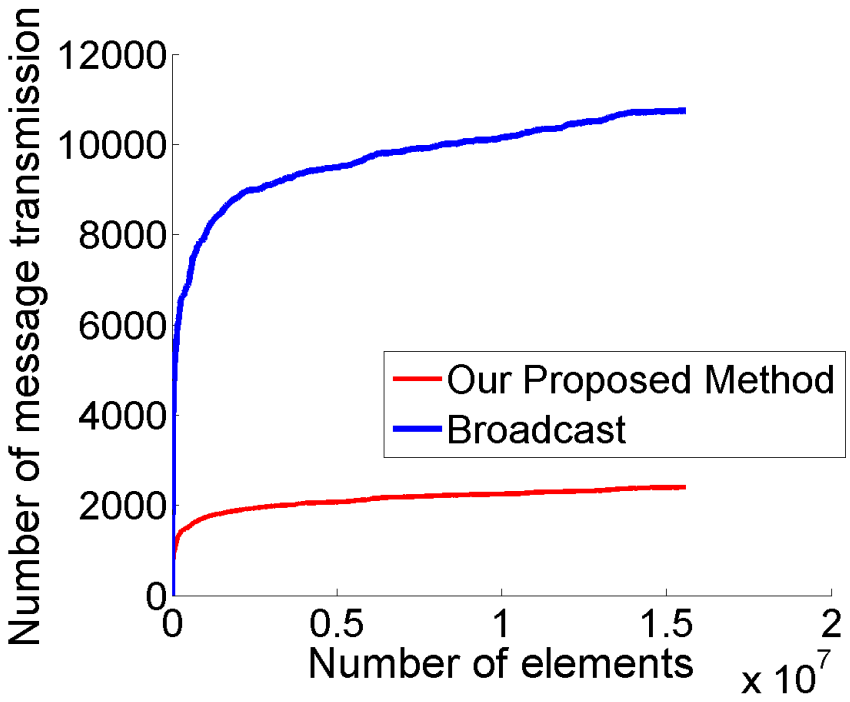


(b) Enron Email Dataset

Figure 5.3 Number of messages as function of the number of sites k.

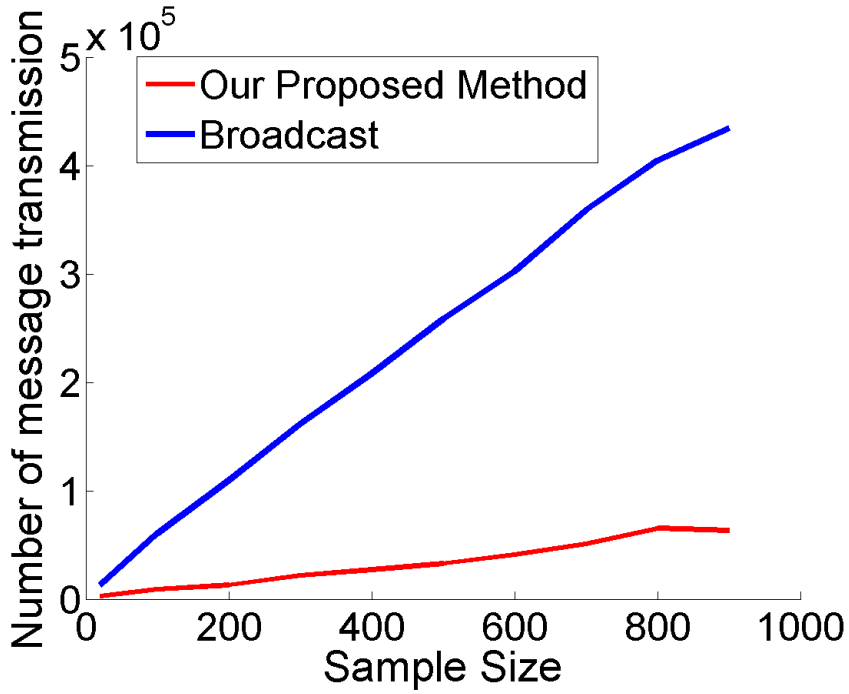


(a) OC48 IP Dataset

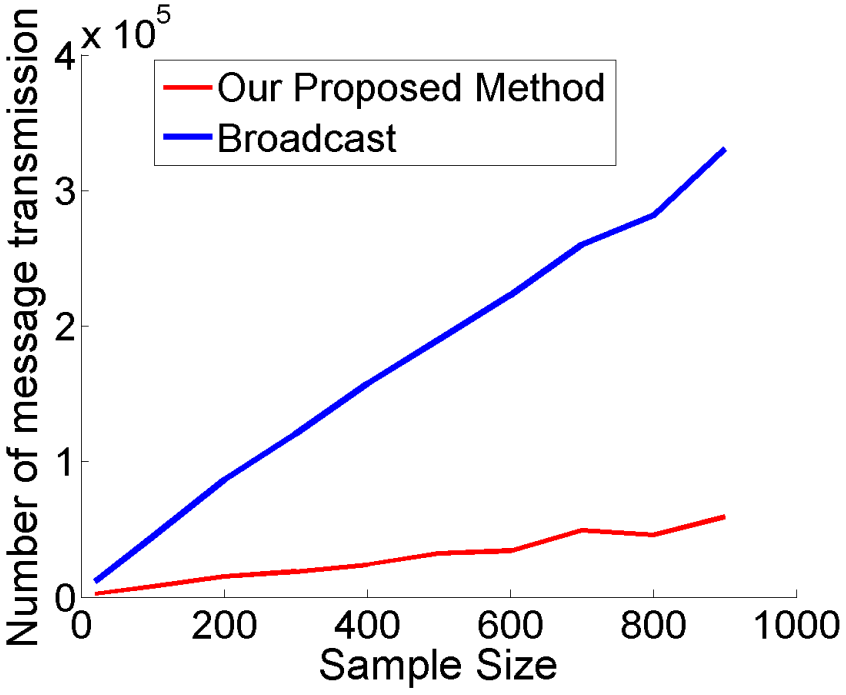


(b) Enron Email Dataset

Figure 5.4 Comparison between number of messages sent by Algorithm Broadcast and our proposed method.

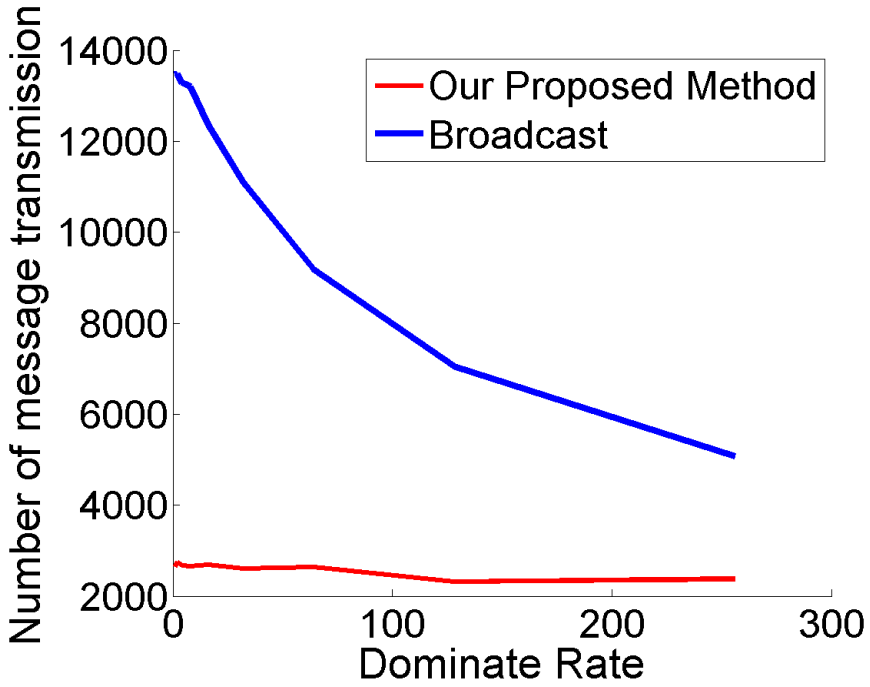


(a) OC48 IP Dataset

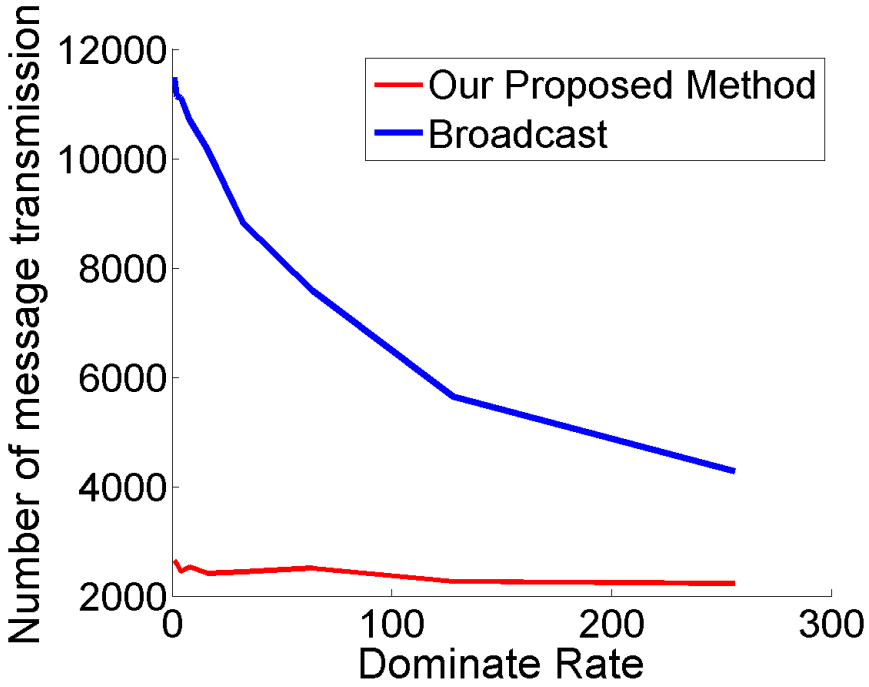


(b) Enron Email Dataset

Figure 5.5 The number of messages sent by Algorithm Broadcast and our proposed method for different sample sizes.

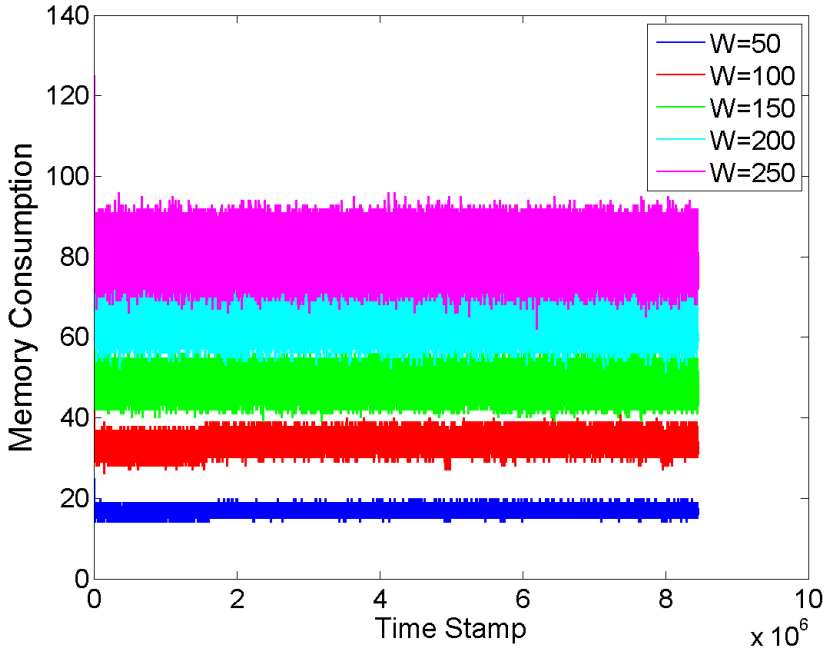


(a) OC48 IP Dataset

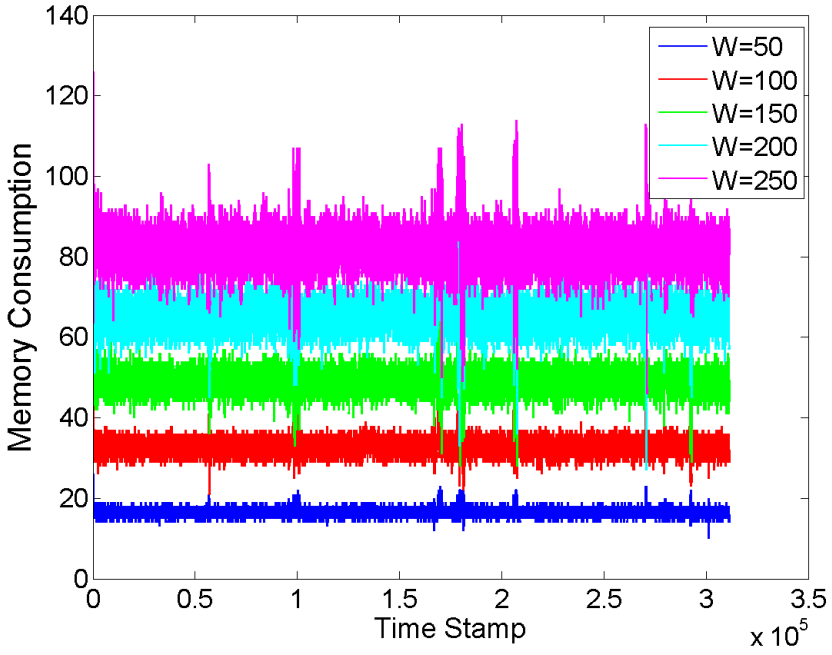


(b) Enron Email Dataset

Figure 5.6 Comparison between Algorithm Broadcast and our proposed method for different dominate rates.

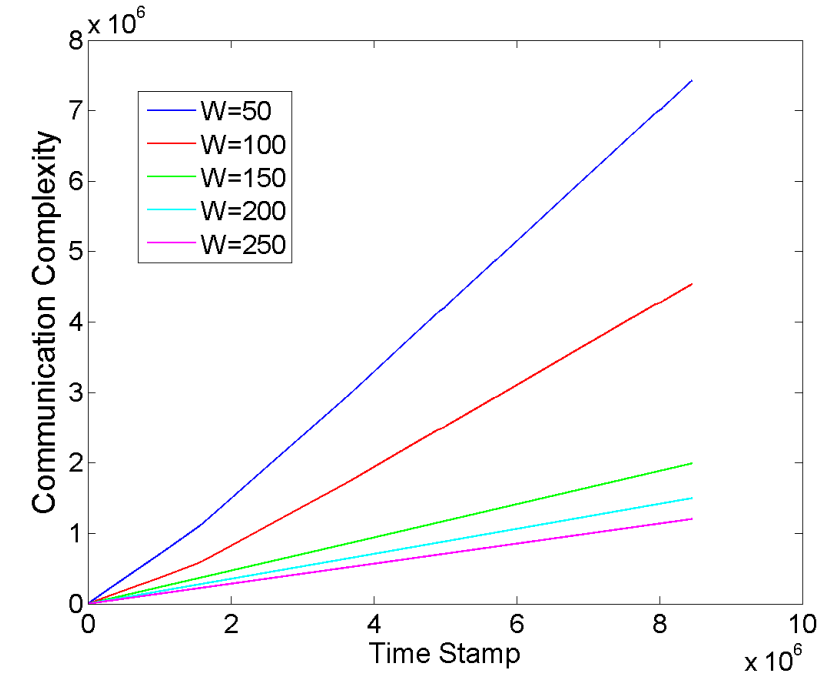


(a) OC48 IP Dataset

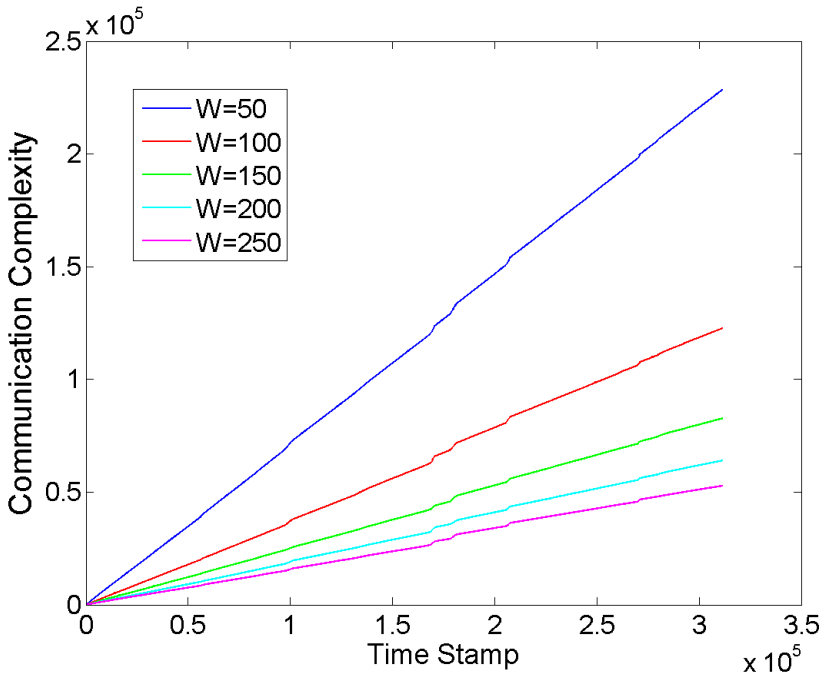


(b) Enron Email Dataset

Figure 5.7 Sliding Windows: Per site memory consumption versus Window Size

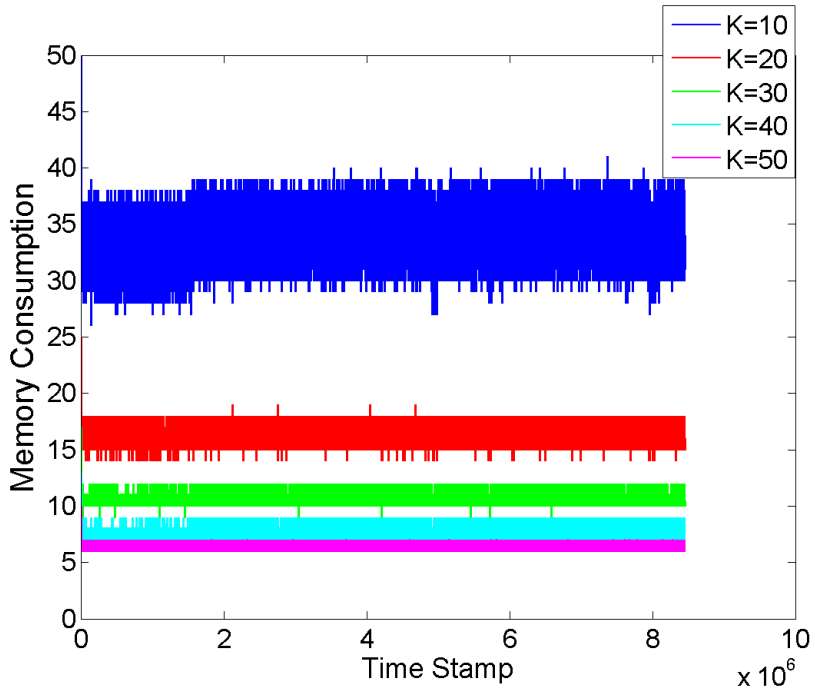


(a) OC48 IP Dataset

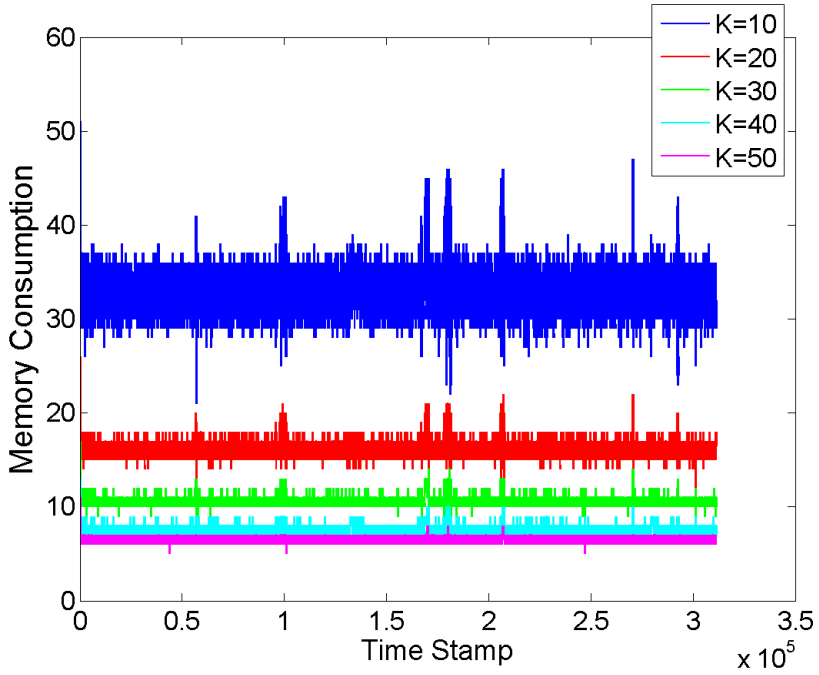


(b) Enron Email Dataset

Figure 5.8 Sliding Windows: Number of Messages versus Window Size

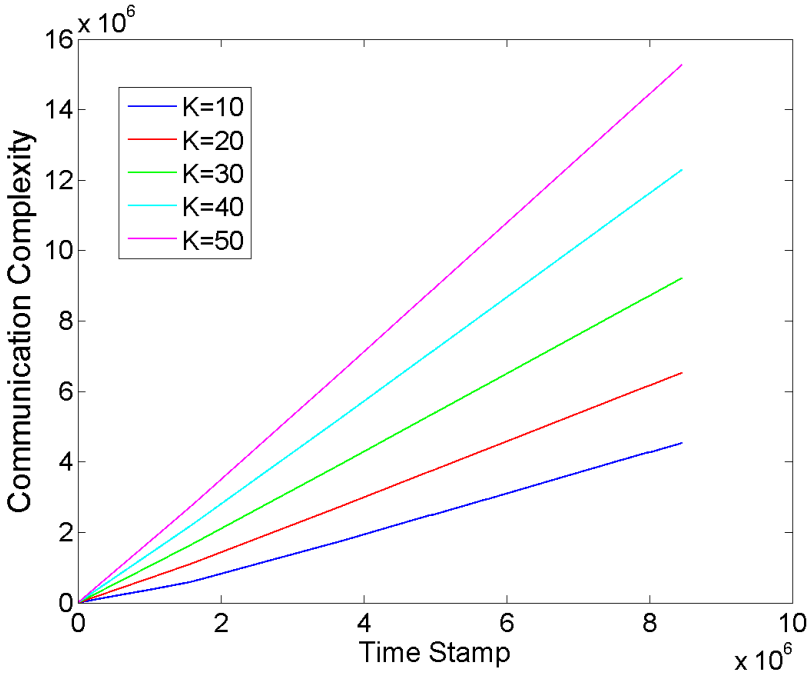


(a) OC48 IP Dataset

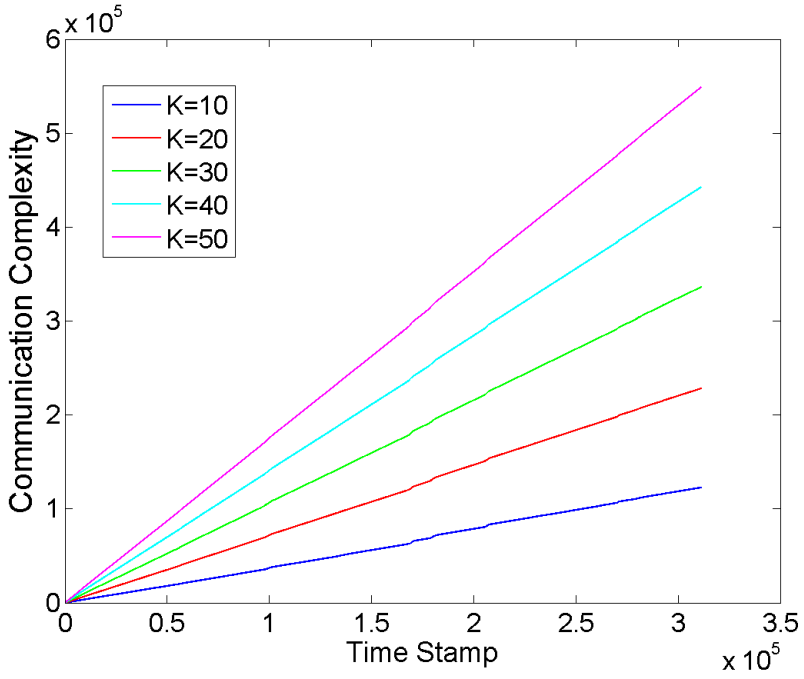


(b) Enron Email Dataset

Figure 5.9 Sliding Windows: per site memory consumption as a function of number of sites



(a) OC48 IP Dataset



(b) Enron Email Dataset

Figure 5.10 Sliding Windows: communication complexity as a function of number of sites

CHAPTER 6. Conclusion

We present new communication-efficient methods for distinct random sampling on distributed data. Our algorithms are practical, easy to implement, and the expected message complexity of the algorithm for infinite window is within a factor of four of the optimal. We present an extension to sliding windows, and an experimental analysis, showing that this method is easily extensible and provides good observed performance.

BIBLIOGRAPHY

Aggarwal, C. (2006). On biased reservoir sampling in the presence of stream evolution. In *VLDB '06*, pages 607–618.

Arackaparambil, C., Brody, J., and Chakrabarti, A. (2009). Functional monitoring without monotonicity. In *ICALP '09*, pages 95–106.

Babcock, B., Datar, M., and Motwani, R. (2002). Sampling from a moving window over streaming data. In *SODA '02*, pages 633–634.

Braverman, V., Ostrovsky, R., and Zaniolo, C. (2012). Optimal sampling from sliding windows. *Journal of Computer and System Sciences*, 78:260 – 272.

CAIDA OC48 Trace Project (2006). The caida ucsd oc48 internet traces dataset - (2002-2003).

CALO Project (2009). Enron email dataset.

Cormode, G. (2013). The continuous distributed monitoring model. *SIGMOD '13*, 42(1):5–14.

Cormode, G., Muthukrishnan, S., and Yi, K. (2008). Algorithms for distributed functional monitoring. In *SODA '08*, pages 1076–1085.

Cormode, G., Muthukrishnan, S., and Yi, K. (2011). Algorithms for distributed functional monitoring. *ACM Trans. Algorithms*, 7(2):21:1–21:20.

Cormode, G., Muthukrishnan, S., Yi, K., and Zhang, Q. (2010). Optimal sampling from distributed streams. In *PODS '10*, pages 77–86, New York, NY, USA. ACM.

Cormode, G., Muthukrishnan, S., Yi, K., and Zhang, Q. (2012). Continuous sampling from distributed streams. *J. ACM*, 59(2).

Datar, M., Gionis, A., Indyk, P., and Motwani, R. (2002). Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813.

Efraimidis, P. and Spirakis, P. G. (2006). Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5):181–185.

Efraimidis, P. and Spirakis, P. G. (2008). Weighted random sampling. In *Encyclopedia of Algorithms*.

Frahling, G., Indyk, P., and Sohler, C. (2005). Sampling in dynamic data streams and applications. In *SCG '05*, pages 142–149, New York, NY, USA. ACM.

Ganguly, S., Garofalakis, M., and Rastogi, R. (2004). Tracking set-expression cardinalities over continuous update streams. *VLDB '04*, 13(4):354–369.

Gemulla, R. and Lehner, W. (2008). Sampling time-based sliding windows in bounded space. In *SIGMOD '08*, pages 379–392.

Giatrakos, N., Deligiannakis, A., Garofalakis, M. N., Sharfman, I., and Schuster, A. (2012). Prediction-based geometric monitoring over distributed data streams. In *SIGMOD '12*, pages 265–276.

Gibbons, P. B. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB '01*, pages 541–550.

Gibbons, P. B. and Tirthapura, S. (2001). Estimating simple functions on the union of data streams. In *SPAA '01*, pages 281–291.

Gibbons, P. B. and Tirthapura, S. (2002). Distributed streams algorithms for sliding windows. In *SPAA '02*, pages 63–72.

Holub, V. Murmur hash 2.0.

Lahiri, B. and Tirthapura, S. (2009). Stream sampling. In *Encyclopedia of Database Systems*, pages 2838–2842.

Seidel, R. and Aragon, C. R. (1996). Randomized search trees. In *Algorithmica*, pages 540–545.

Sharfman, I., Schuster, A., and Keren, D. (2007). A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32(4).

Tirthapura, S. and Woodruff, D. (2011). Optimal random sampling from distributed streams revisited. In *DISC '11*, pages 283–297.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57.

Woodruff, D. P. and Zhang, Q. (2012). Tight bounds for distributed functional monitoring. In *STOC '12*, pages 941–960.

Yi, K. and Zhang, Q. (2013). Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 65(1):206–223.